The Diskless Fileserver

Erik Quanstrom
quanstro@coraid.com

ABSTRACT

The Plan 9 Fileserver is structured as a multilevel cache for direct-attached WORM storage. I describe how the Fileserver is being adapted for modern hardware using network-attached storage (AoE) over 10Gbps Ethernet. This structure allows for good performance and high reliability. In addition it separates storage maintenance from Fileserver maintenance and provides automatic offsite backup without performance penalty.

1. Introduction

In order to meet our growing performance and reliability demands, I am in the process of rolling out a diskless Fileserver. The system consists of a diskless Intel-based Fileserver, a local AoE target and an offset AoE target. A backup Fileserver in "standby" mode is available in case the main Fileserver should fail. The AoE targets are stock *SR1521* machines with added 10Gbps Ethernet cards. This configuration is pictured in Figure 1.

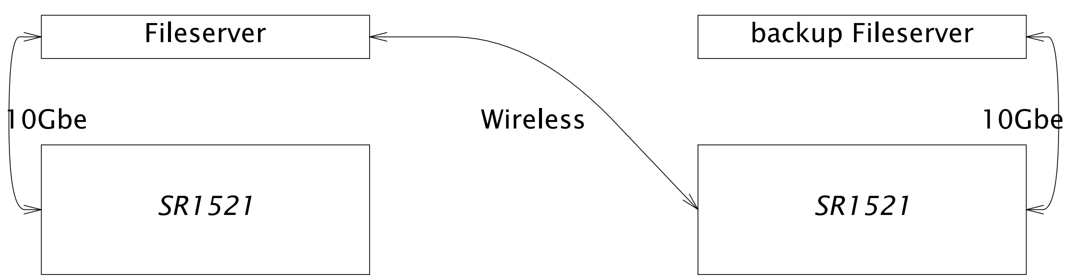


Figure 1

The configuration string[2] for this Fileserver is

```
filsys main ce565.0{e565.1e545.1}.
```

The configuration string for the backup Fileserver is

```
filsys main ce545.0e545.1.
```

The targets `e565.`*x* are connected to the Fileserver by a point-to-point 10Gbps Ethernet link. Except during a dump or in the event of a failure of `e565.1`, all I/O is performed over this link. The target `e545.0` is in another building, connected by a shared 54Mbps wireless link.

The AoE targets are managed independently from the Fileserver. Maintenance tasks,

like replacing failed drives, reconfiguring or adding storage, do not require knowledge of the Fileserver and may be performed without shutting down the Fileserver. Conversely, the Fileserver does not require knowledge of how to perform maintenance on the AoE targets.

2. Fileserver Basics

The Fileserver serves files via the Plan 9 file protocol, 9P2000. Requests that cannot be directly satisfied by the in-memory Block Cache are resolved by devices. The Block Cache is indexed by device and device address. The `cw` device serves the WORM filesystem. It is comprised of three on-disk devices: cache, read-only and cached WORM. These devices are known as `c`, `w`, and `cw`. All Blocks have a `w−address` and a cache state. Blocks not in the cache are state `none`. Freshly written blocks are state `write`. Blocks on the `w` device that are rewritten are state `dirty`. A “dump,” a permanent snapshot of the filesystem, is taken by converting modified blocks to state `dump`. This process takes just a few seconds. Other activity on the Fileserver is halted during the dump. Copying takes place in the background and does not impact the performance of the Fileserver. Once state `dump` blocks are copied to the WORM, their state is changed to `read` or `none`, if it is dropped from the cache. The copying phase of any number of dumps may overlap.

The implemented Fileserver has a Block Cache of 402,197 8192-byte blocks (3137MB), a cache device, `e565.0`, of 3,276,800 blocks (25GB) and a WORM device, `{e565.1e545.1}`, of 1.5TB. The WORM device is the loose mirror of AoE targets `e565.1` and `e545.1`. Writes are preformed on the mirrored devices sequentially but data is read from the first device only. Thus the wireless connection which limits dumps to ~1MB/s is not part of the client’s I/O path.

The WORM filesystem is fully described in [1], [2] and [3].

3. “Standby” Mode

It is not possible use both Fileservers at the same time. Both will try to allocate `w−addresses` without respect to the other. To solve this problem a configuration item and command, both named `dumpctl` were added. The main Fileserver is configured with `dumpctl yes` and the backup Fileserver is configured with `dumpctl no`. To prevent writes, attaches may be disallowed. In the event that the Fileserver fails, the command `dumpctl yes` is executed on the backup Fileserver’s console. And if disabled, attaches are allowed.

While the backup Fileserver is running, it will not see the new data written by the dump process on the main Fileserver. The backup Fileserver must be halted each day after the dump on the Fileserver and the command `recover main` must be typed at the `config` prompt. This will cause the cache to be flushed and the filesystem to be initialized from the new dump.

4. Changed Assumptions

In the fifteen odd years since the Fileserver was developed, a few of its assumptions have ceased to hold. The most obvious is the `worm` device is no longer a WORM. Even if we were to use WORM storage, disk space is inexpensive enough that it would be practical to keep an entire copy of the WORM on magnetic storage for performance reasons. This means that the cache and the WORM devices have the same performance. Therefore it no longer makes sense to copy blocks in state `Cread` to the cache device.

Blocks in state `Cread` have been read from the `worm` device but not modified[3]. A new option, `conf.fastworm`, inhibits copying these blocks to the disk cache.

A less obvious difference is in the structure of the cache. The cache device is structured as a hash table. The hash function is simply modulo the number of hash lines and the lines are written sequentially to disk. If we let n be the number of rows and l be the number of columns in our hash, he function is

```
row = w % rows
c = column + row*n,
```

the blocks will be linearized onto the disk in the following order

$$0, n, 2n, \ldots, (l-1)n, 1, 1+n, 1+2n, \cdots$$

Suppose that two blocks w and $w+1$ are written to the Fileserver with an empty cache. Suppose further that $w+1 \ \% \ l \neq 0$. Then blocks w and $w+1$ map to disk blocks c and $c+CEPERBK$. With a block size of 8192 bytes, current Fileserver parameters and 512 byte disk sectors, this works out to 1072 sectors between "sequential" blocks.

With disk drives of the same era as the original Fileserver, disk transfer rates were limited by hardware buffer sizes and interface bandwidth[5]. Assuming a transfer rate of 1MB/s and a seek time of 15ms, it would take 8ms to transfer 8192 bytes from the disk and less than 15ms to seek to another track or about 347KB/s. On modern SATA drives, it would take 26μs to transfer 8192 bytes from the disk and up to 9ms to seek. this would only yield 890KB/s. During testing about 2MB/s was observed. If this same ratio of calculated versus actual seek time were to hold for older drives, the older drives would operate at near rated bandwidth.

When the formula was changed to

```
n = w % rows
c = column*CEPERBK + n,
```

the blocks are linearized onto disk in the following order

$$0, 1, \ldots, l-1, l, l+1, \cdots ,$$

changing from row- to column-major ordering, performance increased to ~25MB/s. Note that not caching blocks in state `Cread` insures that w and $w+1$ will be stored sequentially on disk, as `column` will be the same for w and $w+1$ unless $w+1 \ \% \ rows = 0$. However, in this case the blocks will also be stored sequentially because row r and row $r+1$ are also sequential.

5. Assumptions Redux

> If a cat can kill a rat in a minute, how long would it be killing 60,000 rats?
> Ah, how long, indeed! My private opinion is that the rats would kill the cat.
> – Lewis Carroll

The Fileserver's read-ahead system consists of a queue of blocks to be read and a set of processes which read them into the cache. Although the original paper on the Fileserver only lists one `rah` process, the earliest Fileserver at the Labs' WORM started four. The Fourth Edition Fileserver again started one `rah` process but attempted to sort the blocks by w—address before processing. This approach probably makes sense on slow, partitioned disks. However, it has the disadvantage of processing blocks serially. The more parallelism one can achieve among or within the Fileserver's devices, the greater the performance penalty of the sequential approach.

To test this idea, a 1GB file was created on the Fileserver on AoE storage. The AoE driver has a maximum of 24 outstanding frames per target. After rebooting the Fileserver to flush the Block Cache, it took 25.5s to read the file. Subsequent reads took and average of 13.72s. After changing the read-ahead algorithm to use 10 independent `rah` processes, the test was rerun. It took 15.74s to read the file. Increasing the number of `rah` processes to 20 reduced the uncached read time to 13.75s, the same as the cached read time. Two concurrent readers can each read the entire file in 15.17s, so the throughput appears to be limited by 9P/IL latency.

6. Core Improvements

The `port` directory underwent some housecleaning. The `9p1` protocol was removed. The console code was rewritten to use the `9p2` code. The time zone code was replaced with the offset pairs from the CPU kernel to allow for arbitrary time zones. A CEC console was added to allow access without a serial console.

More significantly, `Lock`s were changed from queueing locks to spin locks. Since a significant use of spin locks is to lock queues to add work and wake consumers, `unlock` reschedules if the current process no longer holds any locks and has woken processes while it held locks. Also, the scheduler takes care not to preempt a process with locks held. This improved the throughput of single-threaded reads by 25%. These ideas were taken from the CPU kernel.

Networking was changed to allow interfaces with jumbo MTUs. This is not currently used by the IL code as it has no MTU discovery mechanism.

7. PC Architecture Improvements

By far the largest change in the PC architecture was to memory handling. The primary goal was to be able to handle most of the bottom 4GB of memory. Thus the definition of KZERO needed to be changed. The PC port inherited its memory layout from the MIPS port. On the MIPS processor, the high bit indicated kernel mode. Thus Fileserver memory was mapped from 0x80000000 to the top of memory. Converting between a physical and virtual address was done by inverting the high bit. While simple, this scheme allows for a maximum of only 2GB. Lowering KZERO to 0x30000000 and mapping PCI space to 0x20000000 allows for 3328MB memory.

Unfortunately, being able to recognize more memory puts us in greater danger of running into PCI space while sizing memory, so another method is needed. A BIOS 0xe820 scan was chosen. Unfortunately, the processor must be in Real mode to perform the scan and the processor is already in Protected mode when the Fileserver kernel is started. So, Instead of switching back to real mode, `9load` was modified to perform the scan before turning on paging[8].

Surprisingly, the preceding changes were not enough to enable more memory. The Fileserver faulted when building page tables. It turned out this is because the 4MB temporary pagetables built by `9load` were not enough. The BIOS scan of the testing machine yielded 3326MB of accessible memory. This would require 3.25MB of page tables. Since the bottom megabyte of memory is unusable, we don't have any room left for the kernel. The solution was to use 4MB pages. This eliminates the need for page tables, as the 1024-entry page directory has enough space to map 4GB of memory.

On 64-bit processors, it would be relatively easy to fill in more memory from above 4GB by using the 40-bit extensions to 4MB pages.

8. The AoE Driver

> If you were plowing a field what would you rather use, 2 strong oxen or 1024 chickens?
> – Seymour Cray

This is the Fileserver's raison d'être. The AoE driver is based on the Plan 9 driver. It is capable of sending jumbo or standard AoE frames. It allows up to 24 outstanding frames per target. It also allows a many-to-many relationship between local interfaces and target interfaces.

When the AoE driver gets an I/O request, a `Srb` structure is allocated with `mballoc`. Then the request is chopped up into `Frame` structures as available Each is sized to MTU of the chosen link. A link is chosen round-robin fashion first among local interfaces which can see the target and then among the target's MAC addresses. MTUs may be freely mixed. The frames are sent and the number of outstanding frames is appropriately incremented. The driver then sleeps on the `Srb`. When awoken, the process is repeated until all the bytes in the request have been received.

When an AoE frame is received that corresponds to I/O, the frame is copied into the buffer of the `Srb` and the number of outstanding frames is decremented. If there are no outstanding frame remaining, the `Srb` is woken.

Since the Myricom 10Gbe cards have an MTU of 9000 bytes, an entire 8192 byte block and the AoE header fit into a single frame. Thus sequential read performance depends on frame latency. Performance was measured with a process running the following code

```
static void
devcopy(Dcopy *d)
{
	Iobuf *b;

	for(d->p = d->start; d->p < d->lim; d->p++){
		b = getbuf(d->from, d->p, Bread);
		if(b == 0)
			continue;
		putbuf(b);
	}
}
```

The latency for an frame with 8192 data bytes is 79μs giving 12,500pps or 103MB/s while two concurrent reads yield 201MB/s. Testing beyond this level of performance has not been performed.

9. System Performance

I measured both latency and throughput of reading and writing bytes between two processes for a number of different paths. 2007 measurements were made using an *SR1521* AoE target, an Intel Xeon-5000-based cpu server with a 1.6Ghz processor and a Xeon-5000-based Fileserver with a 3.0Ghz processor. 1993 measurements are from [6]. The latency is measured as the round trip time for a byte sent from one process to another and back again. Throughput is measured using 16k writes from one process to another.

Table 1 – Performance				
test	93 throughput MB/s	93 latency μs	07 throughput MB/s	07 latency μs
pipes	8.15	255	2500	19
IL/ether	1.02	1420	78	72
URP/Datakit	0.22	1750	N/A	N/A
Cyclone; AoE	3.2	375	≥250	49

Random I/O was not tested for two reasons. First, ~3GB of recient reads and writes are stored in the Block Cache and when new or newly modified files are reread from the cache, they are reread sequentially. It is expected that the working set of the Fileserver fit in the Block Cache. Second, since a single IL connection is latency limited, reads of highly fragmented files like `/sys/log/auth` from the WORM are not meaningfully slower (59MB/s) than reads from the cache (62MB/s).

10. Discussion

Decoupling storage from the Fileserver with AoE allows for automatic offsite backup, affords good availability, scalablity and performance. The Fileserver is not involved in storage management. It is possible to grow the existing WORM to 9TB without restarting the fileserver. By reconfiguring the Fileserver, essentially unlimited storage may be added.

The size of the WORM and Block Cache have scaled by a factor of 1000 since [3] and single IL connections have scaled by a factor of 200 since [7]. The Block Cache is currently at a practical maximum for a kernel with 32-bit memory addresses. A kernel with 64-bit memory addresses in the next logical step.

The disk cache has not been scaled to the same extent as the increased number of cache buckets put more pressure on the Block Cache and would not provide much benefit. With the `conf.fastworm` option, the cache only need to be large enough to hold the free list and any blocks in state `dirty` or `write`. Eliminating the cache device may make sense in the future. The cache device could be replaced with address of the current Superblock. Addresses below the current Superblock would be read only. The disadvantage to such a scheme is that the dump processes gives the (unused) opportunity to optimize the ordering of `w-addresses`.

11. References

[1]K. Thompson, “The Plan 9 File Server”, Plan 9 Programmer’s Manual, Second Edition, volume 2, AT&T Bell Laboratories, Murry Hill, NJ, 1995.

[2]K. Thompson, G. Collyer, “The 64-Bit Standalone Plan 9 File Server”, Plan 9 Programmer’s Manual, Fourth Edition, volume 2, AT&T Bell Laboratories, Murry Hill, NJ, 2002.

[3]S. Quinlan, “A Cached WORM File System”, *Software — Practice and Experience*, volume 21, number 12, pp. 1289—99.

[4]S. Hopkins, B. Coile, “ATA over Ethernet”, published online at http://www.coraid.com/documents/AoEr10.txt

[5]A. Tanenbaum *Operating Systems, design and implementation*, Prentice Hall, Englewood Cliffs, New Jersey, 1987, p. 272.

[6]Diskless Fileserver source code at /n/sources/contrib/quanstro/src/myfs.

[7]D. Presotto, P. Winterbottom, The Organization of Networks in Plan 9 *Proc. of the Winter 1993 USENIX Conf.*, pp. 271–280, San Diego, CA

[8]Modified 9load source code at /n/sources/contrib/quanstro/src/9loadaoe.